A Minimalistic Approach to End-to-End Protection of Grid Job Payloads

Igor Sfiligoi
Fermilab, Batavia, IL, USA
sfiligoi@fnal.gov

Don Petravick
Fermilab, Batavia, IL, USA
petravic@fnal.gov

Abstract

The security mechanisms for Grid job submission were designed under the assumption that users would submit their jobs directly to remote Grid gatekeepers handling the computing resources. However, in the last several years direct submission has never been the main submission mechanism in Grids like OSG and EGEE, as most users prefer to submit their jobs to a chain of intermediate workload management systems (WMSes) instead. This introduces additional security risks since any WMS can alter the job payload, allowing for execution of arbitrary code in a user's name. In this paper we describe the potential attack vectors and outline a minimalistic end-to-end conceptual solution, based on extensions to user credentials, to contrast them.

1. Introduction

This paper describes a fundamental security risk of the currently deployed Globus GRAM-based Grids, like OSG[1] and EGEE[2], and proposes a minimalistic conceptual end-to-end solution that renders the users' job payloads tamper evident. The proposed solution calls for changes in the endpoints only, allowing for an incremental deployment on the existing Grid infrastructure.

The main part of the paper is contained in the next three sections. Section 2 describes the security risks of the currently deployed infrastructure. Section 3 outlines a high level minimalistic end-to-end proposal for solving the described problems. Section 4 provides a possible deployment scenario using currently deployed tools.

The proposed solution does not attempt to solve all the security problems introduced by a WMS chain and the limitations are described in section 5. A short discussion of related work is presented in section 6 and the proposed future work is presented in section 7.

2. Security risks associated with WMS chains

Computational Grids have become an essential tool of many communities, like the High Energy Particle Physics (HEP) collaborations ATLAS and CMS. This interest has created job handling Grids like OSG and EGEE that are composed of hundreds of independent Grid sites. The sites range in size from tens to many thousands of worker nodes.

The proliferation of Grid sites has made the direct submission of user jobs impractical; selecting the site that will finish a computational job first is far from trivial. For this reason, most organized groups set up a workload management system (WMS) for their users. Example Grid WMSes are glideinWMS[3], the gLite WMS[4] and the OSG ReSS[5].

2.1. Job submission path

Most users submit their jobs to a WMS. To allow a WMS to forward a job to a Grid site, a user has to delegate his/her credentials to the WMS. These delegated credentials will be used by the WMS to submit the user's job to a Grid site, or to another WMS. User's credentials are thus delegated multiple times until the job finally starts. In most cases, the user's job itself will have access to the delegated credentials; they are used by the job for access to non-computational resources, such as storage elements. See Figure 1 for an overview.

The currently deployed infrastructure is based on X.509 proxy certificates[6]. The user will thus use the proxy delegation mechanisms to delegate his/her personal certificate. However, in all currently deployed implementations the delegation is unrestricted; the receiver of the delegated proxy impersonates the user and can effectively perform any action the user can. This requires a strong trust relationship between the user and the credentials' receiver in order not to be abused. There are two issues with this setup: the administrative overhead of

establishing such trust relationship, and the potentially serious consequences associated with a security compromise of the credentials' handing service, like a WMS.

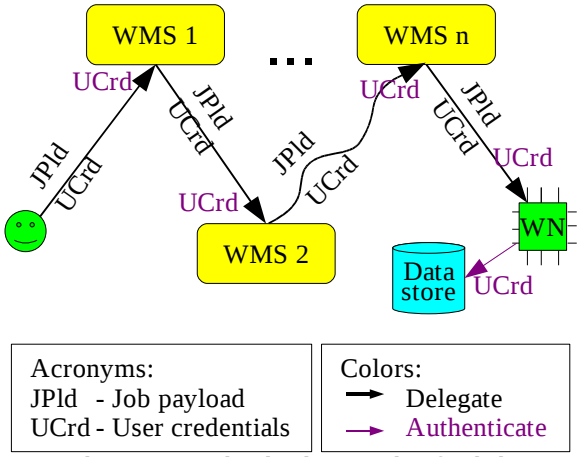

Figure 1. Submission path of a job

Moreover, the delegated proxy certificate does not contain the identities of the entities involved in the delegation chain. If a breach of trust is discovered, it can be very difficult to trace where it originated. Most of the time the only available data are the network related information of the last delegation step, like the protocol used, the IP address and the originating port.

2.2. The malicious WMS use case

This section describes what happens if a job is delegated to a WMS that is not trustworthy. This could happen both because of a security compromise of a trusted WMS and because the WMS was intentionally set up with malicious intents; most of the current Grid information providers do not require an advance establishment of trust to advertise a WMS.

Figure 2 outlines this use case; the malicious WMS can replace the job payload and forward it to the next element in the chain. In Figure 2 the malicious WMS was intentionally positioned in the middle of the delegation chain to clearly illustrate that neither the user nor the worker node may be aware of it. However, any WMS in the chain can abuse the trust, including the first and the last one.

This paper assumes that the end points can be trusted. Users must be trusted, as the Grid services have no way to distinguish between a legitimate and a malicious user; for example, a user cracking encrypted messages on computing resources being granted for genomic research. The worker node must be trusted too; if the worker node wants to run arbitrary code, it can do it at any time, without waiting for a job payload to compromise. The access to non-computing resources is beyond the scope of this paper, and will non be considered in this section. This and other attack vectors not pertinent to this paper are outlined in section 5.

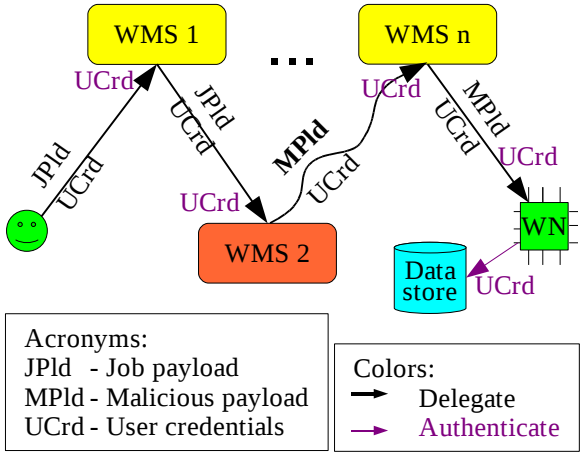

Figure 2. A malicious WMS use case

Let now analyze what is in the job payload, and what harm can come from altering of the job payload. A user will submit his/her jobs using the client tools provided by the WMS of choice. Each tool provides a proprietary job description language (JDL) that the user has to use to transmit the job payload to the WMS. Since no common JDL exist, in this paper we decided to use a minimalistic pseudo-language to describe the job payload. While we do not provide a formal definition of the pseudo-language, the examples are kept simple enough to be understandable. The notation used was selected to be clear and compact.

Job payloads handled by the WMSes in the considered Grids are typically composed of the following four groups of elements:

1) A file to be executed or the name of a file local to the worker node to be executed. Possible examples are a simulation binary executable file, a startup shell script, and the name of the local file, like “/bin/ls”.
2) A (possibly empty) list of command line argument. Each argument is a string and the order of the arguments is important. A possible examples is “-d”,“data.cfg”,”-f”,”log.out”.
3) A (possibly empty) set of environment attributes. Each attribute is composed of a pair containing an attribute name and an attribute

value. The order of the attributes is not important, but all attribute names must be distinct. An example attribute pair is “LD_LIBRARY_PATH”,“./mysubdir:/usr/lib”.

4) A (possibly empty) set of input files. Each input file is composed of a file name (relative to the job startup directory if directory traversal is permitted at all) and file content. The order in which the files are specified is not important, but the name of the files must be distinct. An example input file name is “data.cfg”. A complete example file content is not provided for space reasons, but since it may be useful for discussions, we will use a shorthand like <config data>.

See Figure 3 for an example job payload.

For the purpose of this paper, files staged-in by the user-specified executable after it started running on the worker node are not part of the job payload. It is assumed that the user has applied due care for all the actions of his/her job. Only elements that are transported to the worker node before the job starts are analyzed, because the user has no choice but trust the WMS chain for that task.

```
Executable: <simulation binary>
Cmd args: “-f”, “in.tgz”, “-out”, “out.tgz”
Env args: (“SIM_CONFIG”,”run13.cfg”),
          (“LD_LIBRARY_PATH”,”.:/lib”)
Input files: (“in.tgz”,<input tarball>),
             (“run13.cfg”,<simulation config>),
             (“libsim.so”,<shared library>)
```

Figure 3. An example job payload

Obviously, all of the components of a job payload could be changed by a malicious WMS. The most obvious alteration involves the replacement of the executable. For example, instead of forwarding the user provided simulation executable, a spamming engine could be sent to the worker node, with obvious consequences.

Alteration of command line arguments can also lead to execution of malicious code. An example use case involves the user submitting a job to analyze a virus code. The job payload specifies the local executable “/usr/bin/python” and two command line arguments; “analyze.py” and “virus.py”. If a malicious WMS in the chain inverts the order of the argument, a worker node will execute the virus instead of analyzing it.

Changing the environment attributes or the input files can create similar problems. An example use case involves a user submitting an executable that requires the shared library “sgml-filter.so” and defines the environment attribute “LD_LIBRARY_PATH” to be “/usr/lib/aspell:/lib”. If a malicious WMS in the chain changes the attribute value to be “.:/usr/lib/aspell:/lib” and adds a file with the name “sgml-filter.so” and malicious code content to the job payload, when the worker node runs the specified executable, the malicious shared library is loaded, resulting in running malicious code.

The above mentioned examples clearly show that the current operation mode of the Grid infrastructure poses a real security risk both for the users and for the resource providers. To minimize this risk, the only tool available today consists on building strong trust relationships both between the providers of WMS services and users, and between the providers of WMS services and resource providers.

3. Reducing the risk of WMS chains

Requiring all WMS services to be highly trusted may be difficult to achieve in practice. WMSes are often run by power users in a group of people working on a common project; constant security monitoring and patching is often too big of a burden and is not done. Moreover, Grid WMS software is still an active research area, with many new products being developed and deployed; as with all rapidly evolving products, new security vulnerabilities could get into the code base at every new release.

To make things worse, most WMS installations will handle jobs from hundreds of users. So a single WMS compromise would allow an attacker to manipulate the jobs of hundreds of users. This makes any WMS installation a highly desirable target.

This section proposes a minimalistic end-to-end conceptual solution that will allow for the detection of job payload alterations and thus significantly reducing the risk associated with WMS chains.

3.1. Linking user's credentials to the job payload

Of all the data that the WMSes in the delegation chain handle, only one cannot be altered without detection; the user's delegated credentials. We thus propose that the user embeds a description of the job payload directly in his/her own credentials, before delegating them to any party. This solution is easy to implement and makes the job payload tamper evident.

Since the job specific user credentials[7] are tied to the job payload submitted by the user, if a malicious WMS tries to alter the job payload en-

route, the worker nodes will be able to detect it, and refuse to execute it, as shown in Figure 4.

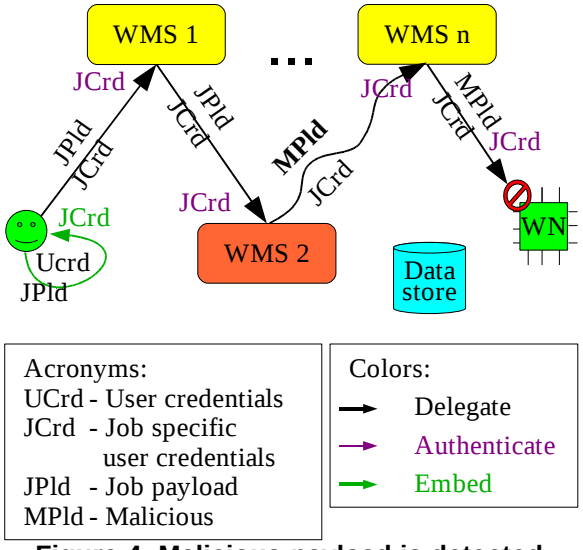

Figure 4. Malicious payload is detected

Note that from a security perspective, only the execution endpoints, i.e. the worker nodes, are required to understand and validate the payload embedded in the delegated credentials, because there is no functional requirement for the WMS to perform any of the job integrity validation. Having to update only the endpoints, i.e. the user submission tools and the WNs, will ease the deployment migration to this new feature.

3.2. Embedding attributes into a X.509 proxy certificate

X.509 proxy certificates support the notion of limiting the proxy certificate by embedding one or more attributes with a set of restrictions. The embedding process uses the original proxy private key to sign the new proxy, thus making the newly embedded attributes tamper-proof. See Figure 5 for an overview.

Each attribute can be labeled either critical or non-critical. Critical attributes technically limit the proxy as they must be interpreted by any proxy handler, with authorization failing if the attribute cannot be interpreted. Non-critical attributes, on the other hand, can be ignored by any proxy handler that does not recognize them, so they are technically just extending the proxy.

Nevertheless, non-critical attributes can be used for authorization purposes by a subset of proxy handlers, effectively limiting the actions of a proxy holder. This is especially useful when the restrictions are not applicable to every possible action a proxy holder can perform, but just to a subset of them.

The job specific user credentials we are proposing fall into this second category; only the the execution endpoints need to understand the attribute containing the job payload description, all other credentials handlers, like WMSes, databases, and storage elements, can safely ignore it. For this reason we are proposing to **embed a non-critical attribute** into the job specific proxy that will be forwarded to the WMSes.

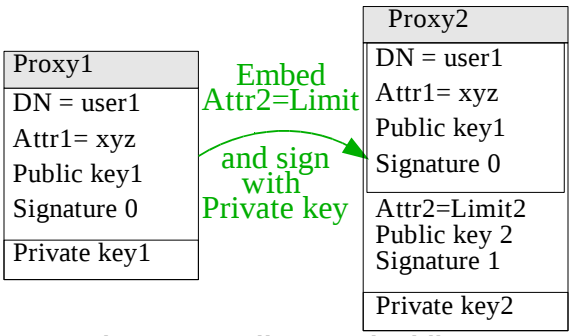

Figure 5. Attribute embedding

3.3. Job payload description

As mentioned in Section 2.2, each WMS client tool uses its own job description language (JDL). It would thus be impractical to embed in the credentials the unmodified job description as specified by the user, because it would require the execution endpoints to understand the JDLs of all the possible submission nodes. For the purposes of this paper, we will thus use a pseudo-language, containing only the strict minimum amount of information needed to achieve our security goals. A real implementation will obviously need a formally defined language, but defining a specific language that would be acceptable for all the endpoint implementations is beyond the scope of this paper.

The job payload description must contain enough information to allow for reliable integrity checking. However, since it will be embedded in the user credentials, whose attributes are world readable, special care must be taken to protect the confidentiality of the job payload; the public part of the proxy should not reveal the content of the job payload. Some communities, like the medical community, have very strict confidentiality and privacy requirements.

It should be noted that the above requirement only protects from deducing the job payload content from

the credentials and does not protect the users from a malicious WMS obtaining the job payload itself; we acknowledge this limitation in section 5 and delegate the solution to this problem to other tools.

A good way to keep the job payload tamper evident while maintaining confidentiality is to use a secure cryptographic hash mechanism, like SHA1. We thus propose to compute the hash values of each and every element of the job payload and use the hash values instead of the original values in the pseudo-description embedded into the job specific user's credentials. The pseudo-description of the example job payload from Figure 3 is shown in Figure 6.

```
Executable: (file,sha1:a1..72)
Cmd args: sha1:9e..87, sha1:3d..a2,
          sha1:c9..ce, sha1:fe..02
Env args: (sha1:d0..92,sha1:26..ab),
          (sha1:95..54,sha1:10..f5)
Input files: (sha1:3d..a2,sha1:fa..23),
             (sha1:26..ab,sha1:01..2f),
             (sha1:13..9c,sha1:7a..12)
```

Figure 6. Pseudo-description of the example job payload

A critical user could observe that hashing every element separately reveals more information about the job payload, like the number of command line arguments and the number of files. It also allows for brute force attacks on the confidentiality of short strings, like is the case for many command line arguments. It also uses up more space compared to hashing the job description as a whole.

We are well aware of these facts, but have chosen this approach for two reasons:

1) It allows for future extensions; if a new attribute type will be added in the future, already deployed execution endpoints can continue to validate the known part, easing the deployment migration.
2) It allows for partial description of the job payload. The user may want to protect the integrity of only the critical elements of the job payload and allow a WMS to change some of the others.

The use case of a WMS changing part of the user's job payload is actually a widely deployed one. For example, the OSG ReSS WMS allows for site specific substitution macros. Passing the name of the storage element nearest to the site is an example valid use case.

3.4. Allowing for partial WMS job payload manipulation

It is conceivable that users may want to use the full WMS matchmaking potential, and for example, ask the WMS to select the appropriate binary for the CPU architecture of the target worker node. The above mentioned ReSS WMS can do this today.

The simple job payload description introduced in the previous section would leave the user in the position of having to choose between security and flexibility; the WMS can only select a binary if the user does not sign the executable element. Having to sacrifice security for flexibility is obviously not desirable.

The above use case can be solved by putting into the job payload description the hash values of all the acceptable executables; it is reasonable to expect that any user will know what those executables are at submit time. A similar reasoning can be applied to all other elements of the job payload.

To better illustrate the concept, let us consider a new use case, by expanding on the original example job. We add the possibility for the job to run on different CPU architectures and different operating systems. The executable and the shared library obviously need to be CPU specific. Moreover, depending if the resource is part of the WMS XYZ group or not, a different configuration file is used.

Figure 7 shows what would be presented to the WMS using our pseudo-language.

```
Executable: on_CPU{x86:<binary1>,
                   SPARC:<binary2>,
                   POWER:<binary3>}
Cmd args: “-f”, “in.tgz”, “-out”, ${jobid}.tgz
Env args: (“SIM_CONFIG”,”run13.cfg”),
          (on_OS{Linux:“LD_LIBRARY_PATH”,
                 AIX:“LIBPATH”},
           ”.:/lib”)
Input files: (“in.tgz”,<input tarball>),
             (“run13.cfg”,if IsXYZ then <config1>,
                                   else <config2>),
             on_OS{x86:(“libsim.so”,<sh.lib.1>),
                   SPARC:(“libsim.sl”,<sh.lib.2>),
                   AIX:(“libsim.o”,<sh.lib.3)}
```

Figure 7. An example dynamic job payload

The associated job payload description will be very similar, but stripped of all the decision semantics; only the available choices are preserved. The reason we do this is because not all directives can be interpreted at the execution endpoints. For

example, the execution endpoints have no way to know if they are part of the XYZ group or not; it is better to just list the available choices. Similarly, semantics of the specific directive for defining the 4th command line argument is WMS specific, so the endpoints cannot validate it. It is better for the user to just blindly trust the WMS and simply not sign that element. We apply the same logic for consistency to all the other elements, although one could argue the OS and CPU checking could be preserved.

The resulting pseudo-description is presented in Figure 8.

```
Executable: oneof{(file,sha1:a1..72),
                  (file,sha1:1f..80),
                  (file,sha1:02..ff)}
Cmd args: sha1:9e..87, sha1:3d..a2,
          sha1:c9..ce, any
Env args: (sha1:d0..92,sha1:26..ab),
          (oneof{sha1:95..54,sha1:9e..bc},
           sha1:10..f5)
Input files: (sha1:3d..a2,sha1:fa..23),
              (sha1:26..ab,oneof{sha1:01..2f,
                                 sha1:f9..8a}),
              oneof{(sha1:13..9c,sha1:7a..12),
                    (sha1:99..c8,sha1:f1..23),
                    (sha1:18..1b,sha1:a2..88)}
```

Figure 8. Pseudo-description of the example dynamic job payload

4. An example of a possible real world deployment scenario

To allow the reader a better understanding of the conceptual proposal outlined in section 3, this section describes a possible real world deployment scenario. The presented example includes possible extensions to existing tools; we cannot and we do not claim that this will be the actually implemented solution.

The deployment scenario we are presenting is based on the OSG Grid and using glideinWMS WMS. This choice is due solely to the fact that the authors are very familiar with this setup. Examples with other Grids and other WMSes would be equally representative.

4.1. The glideinWMS in OSG

The glideinWMS[3] is a pilot-based WMS based on the Condor batch system[8]:

- The glideinWMS is defined by a **condor_collector,** an information collection process.
- Users use the **condor_submit** client tool to submit their jobs to one of the **condor_schedd** processes; each **condor_schedd** holds a job queue and handles the received user payloads.
- The computing resources are gathered asynchronously by means of **pilot jobs**; the glideinWMS uses Condor-G to submit pilot jobs to various Grid sites, using special pilot credentials. Each pilot job contains a **condor_startd** daemon; when a pilot job starts on a worker node, it contacts the **condor_collector**.
- After a matchmaking process, the **condor_schedd** delivers a user job to the **condor_startd**.
- The **condor_startd** invokes the local **gLExec**[9] privileged executable to identify the user and run the job.

In this scenario, the only fully trusted elements are the endpoint nodes; software wise, the **condor_submit** client tool on the submit side, since it is under user control for the whole duration of the job submission, and the **gLExec** privileged executable on the execution side, since it is installed and maintained by the worker node administrator. These are thus the only pieces that need to be changed in order to guarantee end-to-end integrity of the job payloads.

4.2. Changes to condor_submit

Today, **condor_submit** parses a Condor submit file and uses it to create a job payload description using the ClassAd language. It then uses the user proxy to authenticate to a **condor_schedd**; after the connection has been established, it sends over the job ClassAd, the input files and finally delegates the user proxy. The **condor_schedd** returns a job number and **condor_submit** terminates; the user can now destroy the proxy used by **condor_submit**. However, the delegated proxy and the job payload are held by the **condor_schedd** for an extended period of time; a compromise allows an attacker to replace the user job payload with a malicious one at any time.

To implement the security feature described in section 3, the **condor_submit** command needs to be changed so that after parsing the submit file, it first creates a new job specific user proxy; all the elements that are fully specified by the user are converted in hash values and embedded into the proxy. The newly created job specific user proxy is then used for authentication and is also the one being

delegated to the **condor_schedd**; the original (unrestricted) user proxy is not used anymore, and could be destroyed immediately after the job specific one was created.

4.3. Changes to gLExec

Today, **gLExec** is the site-trusted custodian of a worker node for the pilot-based jobs. A user is only held responsible for the code running on the worker node, if **gLExec** itself started it. **gLExec** is a privileged executable that expects a user proxy, a user binary and the user command line arguments as its input; once the user is authenticated and authorized, it cleans the environment, switches to the appropriate local account, copies the user proxy in a location readable only by the local account, and executes the provided user binary and arguments.

As you may notice, the currently deployed **gLExec** does not have any support for handling user specified environment attributes nor for user provided input files. The **condor_startd** gets around this limitation by submitting a Condor-specific wrapper script that does the trick. Exact details are beyond the scope of this paper, but it is clear that the currently deployed **gLExec** allows for execution of code that was not provided by the user; if the wrapper script ran malicious code, the user would be held responsible.

To implement the security feature described in section 3, gLExec (or equivalent tool) must be:

1) Extended to support natively at least the four job payload groups listed in section 2.2[1].
2) Extended to understand the new proxy attribute and perform the needed integrity checks.

Once all the job payload is exposed to **gLExec**, it can compute the hash values of all the elements and compare them to the job description embedded in the job specific user proxy; if even one of them does not match, **gLExec** will refuse the request to run the job. The job payload tampering by any process of the glideinWMS will thus result only in the job not being run. This also implies that the pilot can't use any wrapper around the user jobs.

5. Limits of the current proposal

Our proposal does not claim to solve all the security problems introduced by the introduction of a WMS chain in the jobs submission workflow. Indeed, it only addresses the problem of accountability of job payloads executing on trusted nodes.

To the best of our knowledge, the other problems introduced by the use of WMS chains, and not covered by out proposal, are:

1) A malicious WMS can decide not to protect the confidentiality of the job payload.
2) A malicious WMS can alter the output sandbox of a finished job, returning to the user a modified result.
3) A malicious WMS can use the user's credentials to access non-computing resources, like storage elements and databases. The job specific user credentials are as permissive as the current (unrestricted) user credentials.
4) A malicious WMS can accept a job and never forward it to any computing resource.
5) A malicious WMS can forward multiple copies of the same job.
6) A malicious WMS can forward a job to computing resources that the user explicitly said should not be used.

Problem #1 affects the confidentiality of the job payload. Problems #2 and #3 affect the integrity and confidentiality of user's data. Problems #4, #5 and #6 affect the availability of computing resources. While we fully acknowledge that all of the above are important problems, our intention was to provide a minimalistic solution that protects the job payloads end-to-end and that is easy to implement with minimal changes to the currently deployed infrastructure.

6. Related work

The risk of the delegation of unrestricted users credentials have been recognized contextually with the introduction of the X.509 proxy certificates. However, the only protection in use today is the lifetime restriction; proxy certificates are supposed to be short lived, in order to minimize in the time dimension the amount of damage a stolen proxy can do. Unfortunately even this is difficult to use in practice, as it is not unusual for a job to take several days, or even weeks, from submission to completion.

Task-specific proxies have been introduced in the context of Condor[7] by I. D. Alderman and M. Livny. While this work is closely related to that, we downsized that proposal to the bare minimum requirements needed to ensure end-to-end protection of the job payloads in the analyzed Grids. By providing a minimalistic proposal, we hope to

1 Requirement #1 will require also the change to the pilot infrastructure, but this is just due to the current design decisions of gLExec, and is not directly security driven.

provide a base discussion platform involving all current endpoint providers, that will result in a deployable system in the near future.

D. Snelling et. al. [10] have proposed in the context of UNICORE a mechanism similar to ours that is also based on integrity checks of job payload. While similar, it is UNICORE specific and not directly portable to GRAM-based Grids, like OSG and EGEE.

7. Future work

This paper has presented only a conceptual solution to the problem of end-to-end protection of the job payloads. In particular, we used a pseudo-language to describe the job payload. In order to implement such a solution, a formal language must be selected, either among existing ones or by defining a new one.

If our proposal is well received by the Grid community, we plan to work with the interested parties on the selection of such a language. We will also select and register an appropriate X.509 OID to hold the job description. The results should be submitted to a standards body to facilitate full interoperability across a wide range of implementations.

Furthermore, since the authors are involved in the development of pieces of Grid middleware, we will work on the implementation of the emerged standard in our products.

8. Conclusions

The introduction of WMS chains in the Grid job submission workflow has introduced additional security risks. One of the major risks comes from the possibility of any WMS in the chain to alter any user's job payload, possibly resulting in the execution of malicious code in the user's name.

In this paper we examined the possible abuse cases and presented a conceptual solution based on job specific user credentials. The proposed solution could be easily implemented by modifying the trusted endpoints only, allowing for a staged deployment.

We acknowledge that the proposed solution will not solve all the security risks introduced by the use of a WMS chain for job submission, but we decided to opt for a minimalistic solution that is easy to implement but still remediates an important security vulnerability.

It is worth noting that while this paper worked within the context of Globus GRAM-based job processing Grids, like OSG and EGEE, most of the high level concepts presented should apply to any job processing Grid.

9. Acknowledgments

Fermilab is operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

The authors thank Ian D. Alderman and Frank Siebenlist for all the good ideas that helped shape this paper.

10. References

[1] Ruth Pordes, et. al., “The open science grid”, *Journal of Physics: Conference Series* ***78*** , Institute of Physics Publishing, 2007 (15pp),

[2] Home page of the EGEE project, http://www.eu-egee.org, Accessed April 2008.

[3] I. Sfiligoi, “ Making science in the Grid world: using glideins to maximize scientific output”, *Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE* ***2***, Honolulu, HI, USA, 2007, pp. 1107-1109.

[4] P. Andreetto et. al., “Practical approaches to grid workload and resource management in the EGEE project”,. *Proceedings of CHEP’04*, Interlaken, Switzerland, 2004.

[5] “OSG Resource Selection Activity”, https://twiki. grid.iu.edu/twiki/bin/view/ResourceSelection/WebHome, Accessed April 2008.

[6] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson, “Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile”, *RFC 3820*, 2004.

[7] I. D. Alderman and M. Livny, “Task-specific restricted delegation”, *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, Monterey, CA , USA, 2007, pp. 243-244.

[8] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience" *Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4*, 2005, pp 323-356.

[9] D. Groep, O. Koeroo, G. Venekamp, “David Groep, Oscar Koeroo, Gerben Venekamp”, *To be published in Journal of Physics: Conference Series (JPCS) CHEP2007*, Preprint: http://www.nikhef.nl/grid/lcaslcmaps/glexec/glexec-chep2007-limited.pdf

[10] D. Snelling, S. Van Den Berghe, V. Li, “Explicit Trust Delegation: Security for Dynamic Grids”, *Fujitsu Sci Tech J Vol 40 No 2*, 2004, pp. 282-294.